VLSI DESIGN
1996, Vol. 4, No. 1, pp. 17-32
Reprints available directly from the publisher
Photocopying permitted by license only

Nearly Balanced Quad List Quad Tree - A Data Structure for VLSI Layout Systems*

PEI-YUNG HSIAO
Department of Computer and Information Science, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu, Taiwan 300, R.O.C., Phone: 886-35-712121ext56630, FAX: 886-35-721490, E-mail: pyhsiao@cis.nctu.edu.tw

(Received October 4, 1993, Revised December 24, 1994)

In the past ten years, many researchers have focused attention on developing better data structures for storing graphical information. Among the proposed data structures, the *quad tree* data structure provides a good way to organize objects on a 2-D plane. Region searches proceed at logarithmic speeds a desirable characteristic, but no previously proposed VLSI quad tree data structure distributed objects to subdivide the spatial area. This has been a major drawback for operations such as *tree searching* and *window query*. In this paper, we present a new division method to reconstruct those quad trees including the *multiple storage quad tree* (MSQT) and the *quad list quad tree* (QLQT) into nearly balanced *quad tree* data structures. Nearly balanced quad trees based on our new spatial division method are constructed by dynamically translating unbalanced multiple storage quad trees or unbalanced quad list quad trees into balanced structures. All benefits of the original quad tree data structures are completely retained. In addition, this method is simple and balanced quad trees memory require less than the original quad trees. Experimental results illustrate that the improvement in region queries of the presented nearly balanced quad trees to both of the QLQT and the MSQT is better than the improvement of the QLQT to the MSQT.

Key Words: *VLSI Layout Systems, Region Query, Quad Trees, Corner Stitching, Spatial Data Structure*

1 INTRODUCTION

In interactive CAD applications, a suitable data structure for the storage of graphical information is the most important factor in determining the overall performance of the application. We have to take the speed of key operations, such as windowing, tree traversal, and neighborhood searches, as well as the available memory resources into consideration. Thus, many storage methods ranging from simple linear lists to more sophisticated data structures, such as corner-stitching [1], large k-d tree families [2]–[6], and quad trees [7]–[13], have been introduced to enhance the application of VLSI-CAD tools [14]–[20].

In 1974, Finkel and Bentley proposed a quad tree data structure for organizing points on a plane so that one can quickly find the points in any specified query region [8]. In 1982, Kedem modified this quad tree data structure to allow extended objects such as boxes and polygons to be retrieved by localization [9]. Indeed, quad tree data structures are an excellent way of organizing objects on a plane. A quad tree, as indicated by its name, repeatedly divides the layout space into quadrants [8]. This subdivision generally continues until the quadrants are small enough so that each contains only a few objects.

Brown's multiple storage quad tree (MSQT) [11], was proposed in 1986; in 1989, a more efficient data structure, the quad list quad tree (QLQT) [12], was introduced. Both of these quad tree data structures are constructed by regularly dividing a quadrant recursively into four subquadrants of equal area. Division procedures used in the MSQT and QLQT do not consider the distribution of objects in the 2D plane. The QLQT improves the MSQT in the leaf quadrants by transforming its single long list of object reference nodes into four exclusive lists[12].

The QLQT features single-access to any object while region query is running. In other words, it does not need the marking and unmarking operations that multiple-storage quad trees use for any visited object. As a result, their experimental results (only about 7000 objects in the

*This work supported in part by NSC 82-0404-E004-129, Rep. of China

layout plane were considered) showed that the speed of region queries for large windows in QLQT is 2 to 5 times faster than those in the MSQT. However, it is not actually necessary to unmark each MSQT object description node's flag for every region query. The marking and unmarking of visited objects proposed by Brown merely avoids reporting any object more than once while querying in the MSQT. Therefore, we have modified Brown's method to speed up region queries in the MSQT by the small improvement of discarding the unmarking operation and replacing local flags with global ones. Our experimental results, including these from three kinds of problem size, 1000, 10000 and 70000 objects in the layout plane, show that improving MSQT marking and unmarking operations did indeed speed up the region query function.

However, when we examine the division method, as shown in Figure 1, we see that both types of tree have unbalanced structures. This shortcoming significantly reduces search and window query efficiency, especially in the worst cases. Generally speaking, quad tree data structures must be balanced if the speed of the region search is to be increased. VLSI layout tools based on balanced data structures will certainly benefit from this advantage since they perform a lot of window query operations.

It should be mentioned here that Su *et al* [29] and De Pauw [30] had proposed ways of improving the performance of MSQT and QLQT. Su *et al.* [29] suggested an adaptive threshold number and a new quad subdivision criterion to take care of nonuniform distributions of object positions and object sizes. De Pauw [30] proposed a *multitree with internal storage* (MTIS) structure which basically stores the object reference pointers at the internal nodes rather than the leaf nodes of the tree structure. By moving the object reference pointers nearer to the tree root, this method speeds up the region search operation. Nevertheless, neither of these two data structures considers tree balance.

In the practical VLSI tool design applications, the number of tree-building operation is much smaller than the number of region queries (the ratio may be 1 to 100 or 10 to 1000). The overhead of reconstructing the original quad trees into nearly balanced quad trees can therefore be ignored, especially since the proposed binary division method illustrated in Sec. 3 is so simple.

This paper presents two nearly balanced quad trees called the *binary balanced multiple storage quad tree* (BBMSQT) and *binary balanced quad list quad tree* (BBQLQT). According to our experience, the BBMSQT and the BBQLQT are absolutely balanced for all general cases of layout distribution except for the worst cases discussed in Sec.5. Even for the worst cases, our nearly balanced quad trees still preserve the advantages of requiring less memory and conducting region query operations quickly.

2 QUAD TREES FOR LAYOUT REPRESENTATIONS

2.1 The Basic Quad Trees

A quad tree, as shown in Figure 1, is constructed by repeatedly dividing all quadrants containing more than T_N objects (T_N is a threshold number introduced by Brown [11]) into four subquads. This is done by associating a tree node with each quadrant and drawing four links from each tree node to its four child nodes [8][11], T_N thus varies according to the total number of objects.

2.2 Bisector List Quad Trees (BLQT)

In Kedem's bisector list quad trees [9][11], each non-leaf quadrant is associated with a pair of horizontal and vertical bisector lists. Each object that is not entirely contained in a single subquadrant of a non-leaf quadrant is put in either its horizontal or vertical bisector list. Window-finding operations can be very inefficient when using the bisector list quad tree data structure especially when the window is small and the mask is large. This is because the window-finding operations have to examine objects organized into linear lists. This linear list search is complicated and greatly reduces the efficiency of the window-finding operations.

2.3 Multiple Storage Quad Trees (MSQT)

In 1986, Brown [11] proposed another way of organizing objects that intersect one or more than one quadrant boundary. Such objects are stored in every leaf quadrant they intersect, which means that some objects will be stored in more than one quadrant. This approach wastes some space storing duplicated pointers, since some objects are referenced in more than one quadrant. In order to avoid reporting objects more than once, other authors presented further improvements [12][14]. Objects are marked the first time they are reported, and once marked, are not reported again. This data structure requires marking and unmarking operations to maintain its validity whenever the objects are sought.

2.4 Quad List Quad Trees (QLQT)

In response to the problem raised by multiple-object enumeration in an MSQT, the quad list quad tree data

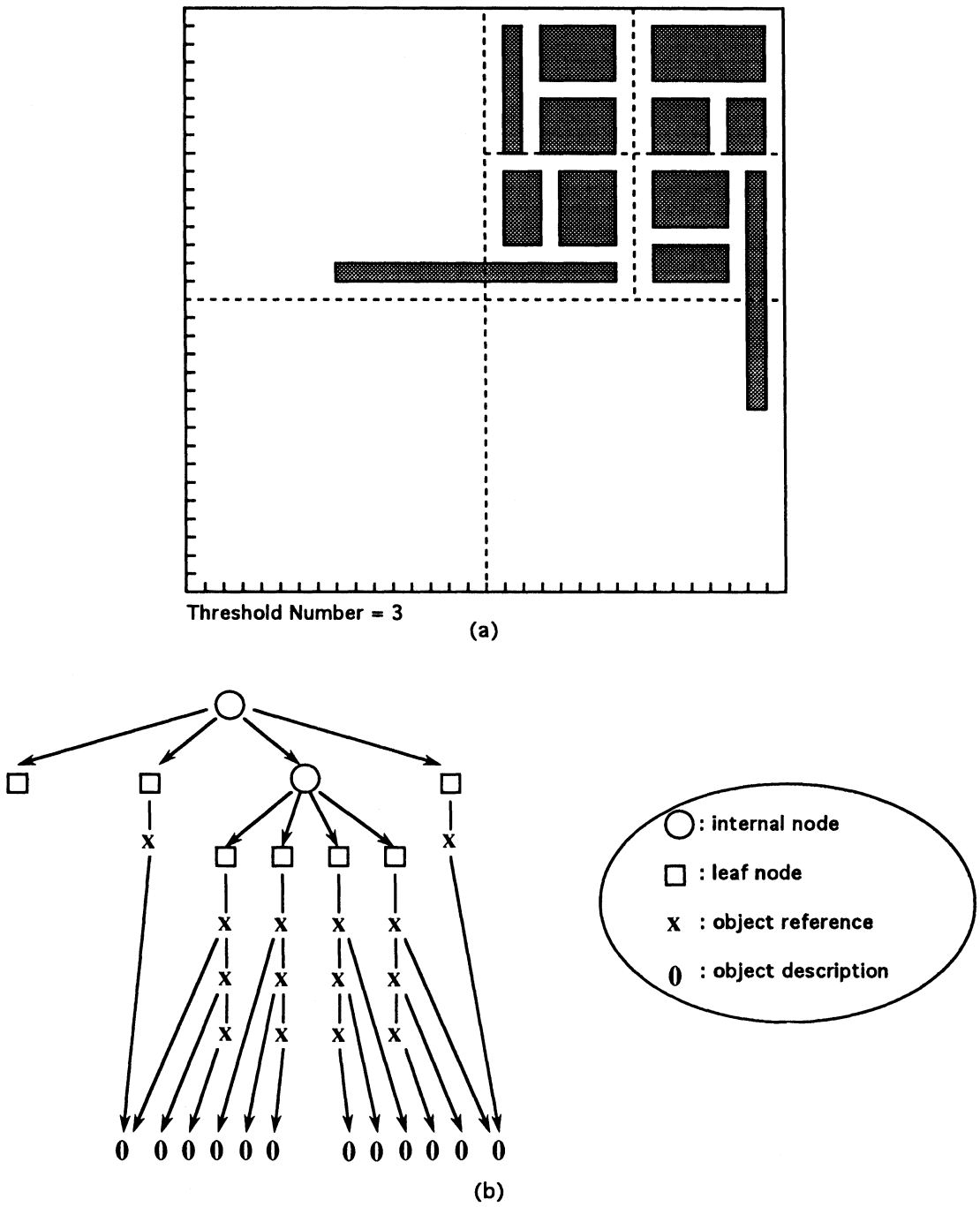

FIGURE 1 The division method used by MSQT and QLQT. (a) A set of 12 rectangles on the layout plane with 32*32 square units. (b) The corresponding MS quad tree.

structure was presented in 1989 [12]. In a QLQT, the single long linked list of object reference nodes is made into four distinct shorter lists. If any object intersects the leaf quadrant, a reference to this object will be included in one of the four lists according to the relative position of the object with respect to the leaf quadrant it intersects. The assignment procedure [12] is illustrated in Figure 2 and Table 1, where four different lists (0 to 3) are associated with each leaf node. If the object's smallest enclosing rectangle overlaps the subregion (Figure 1) corresponding to this leaf node, a reference to this object will be included in one of the four lists depending on its relative position with respect to the subregions shown in Table 1. When objects are searched for or found

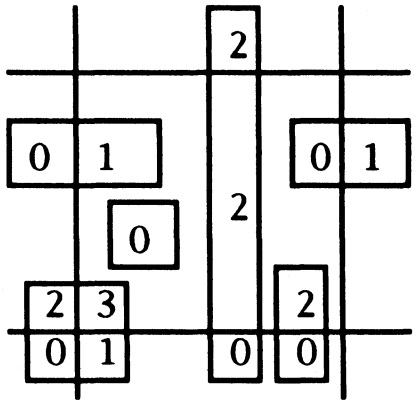

FIGURE 2 The list codes for object references depending on the object position within the quadrant.

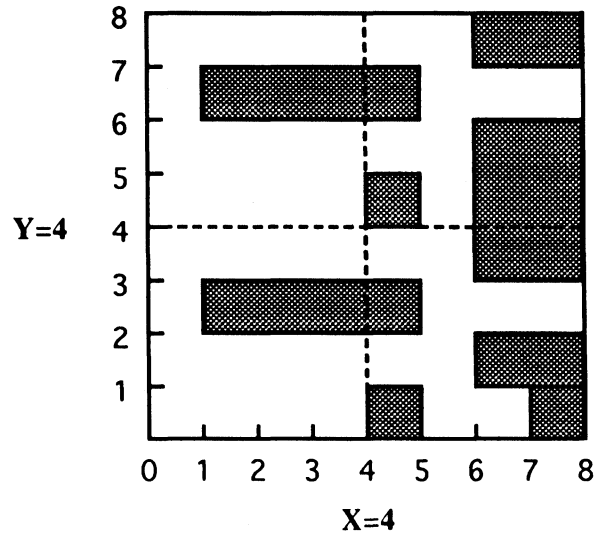

FIGURE 3 The two orthogonal dashed lines, X=4 and Y=4, recursively dividing the layoút plane into quadrants of equal size.

in the QLQT data structure, each object will be accessed only once from one of the four distinct lists instead of from the longer list in the MSQT. Even so, both the MSQT and the QLQT form skewed (heavily unbalanced) trees and require a linear time but not in O(log N) to perform search operations, where N is the number of objects in the layout plane.

3 THE NEW BALANCED QUAD TREES (BQT)

3.1 The Original Division Method

In Brown's MSQT [11] and Weyten's QLQT [12], the division method used to construct the entire quad tree data structure involves simply splitting each quadrant that contains more than T_N objects into four smaller subquadrants of equal size, as shown in Figure 3. Each parent quadrant is represented by a tree node that points to the four subquadrants. This process recursively subdivides any of new subquadrants that contain more than T_N objects.

This division method is suitable for a uniformly distributed layout, but unfortunately layout objects in the real world are usually not distributed in a completely uniform manner. Because, in most physical layout systems, the system global layout plane, say from, $(-2^{31}, -2^{31})$ to $(+2^{31}, +2^{31})$ is usually much larger than the realistic mask layout. Moreover, in the worst cases, these data structures lead to a linear time complexity rather than the expected time order of O(log N) for the tree search operations (N is the total number of objects in the layout plane).

TABLE 1
The Coding of List Type

rectangle crossing lower boundary	rectangle crossing left boundary	list type
False	False	0
False	True	1
True	False	2
True	True	3

3.2 The New Division Method to Guarantee a Nearly Balanced Tree

In our approach, we improve the division method by taking the distribution of layout objects into consideration so that each of the four subquads contains about the same number of objects. Hence the constructed quad tree data structure will be a nearly balanced tree except for the worst cases discussed in Sec.5.

In this method, the quadrant to be split is divided by two orthogonal dividing lines. However, each of the resulting four subquads may have a different area. In other words, we manipulate the final position of the cross point of the two orthogonal dividing lines to give the four subquads about the same number of objects. This is illustrated in Figure 4.

In adjusting the position of the cross point, we can separate the whole division procedure into two independent subprocedures, which means we have to locate the vertical line and horizontal line individually in order to locate the new cross point, as illustrated in Figure 5. Often this division method does not produce a fully balanced quad tree which guarantees the order of tree height in $\log_4 N$. However, in the worst case, the quad trees produced by our division method will guarantee the tree height limited at $\log_2 N$. The independent consideration in each orthogonal direction to locate the two

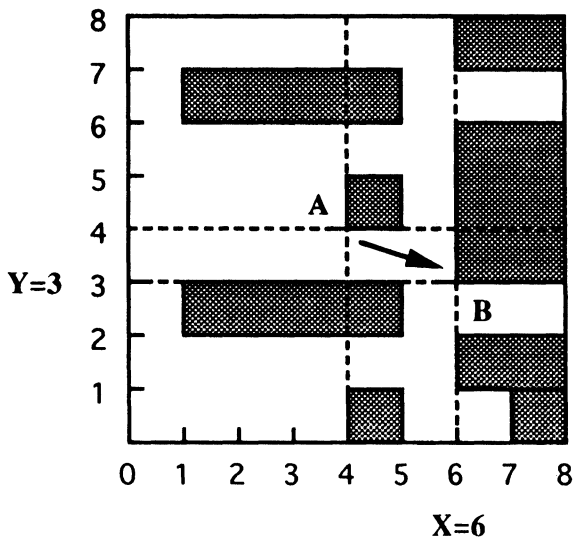

FIGURE 4 To adjust the cross point from A to B. The orthogonal dividing lines, X=6 and Y=3, split the entire region into four subregions where each subregion contains two objects.

dividing lines is not yet an optimized global consideration and does not always obtain a quad balanced quad tree. Hence, the constructed quad trees based on the binary division approach as shown in Figure 5 are called binary balanced quad trees.

In determining the locations of the dividing lines, we can use a binary search method, as shown in Figure 6 to adjust the lines in both the vertical and horizontal directions to obtain the near balanced position repetitively. We refer to the original unbalanced quad tree (the MSQT or the QLQT) and repeatedly examine the differences between the number of objects contained in each candidate subregion to determine the new candidate dividing position; this process continues until an exact line location is determined. In counting the number of objects, we use the region query function from the original quad tree.

This division process recursively subdivides any new subquads containing more than T_N objects. Finally, the balanced quad tree is constructed by translating the original one. Whenever the quad tree becomes somewhat unbalanced, which frequently occurs after a sequence of insertions or deletions, the tree will be dynamically rebalanced by the above procedures. Figure 7 shows balanced quad tree constructed by translating the original quad tree shown in Figure 1 using the proposed division method.

4 THE TRANSLATING ALGORITHMS

In this section we introduce the detailed algorithms for translating the MSQT into a balanced quad tree. These algorithms can easily be modified to fit any quad tree data structure that employs the region query function.

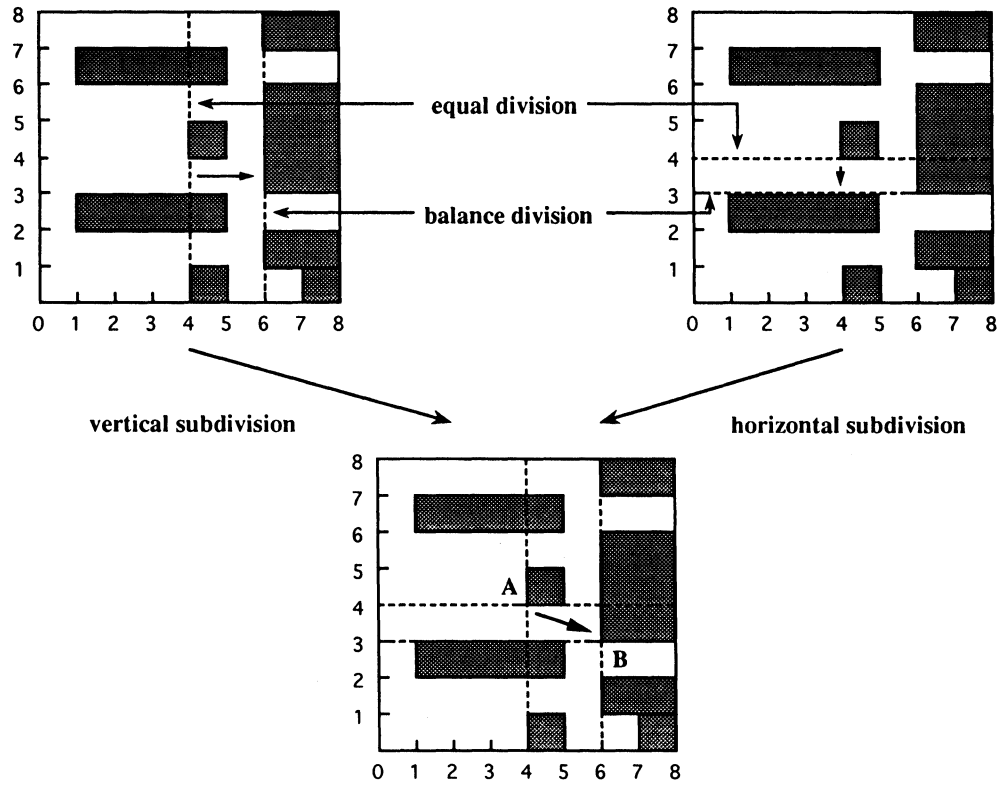

FIGURE 5 To adjust the cross point from A to B, the entire procedure can be divided into two individual sub-procedures which are independent of each other. After the two orthogonal dividing lines are decided independently, we can then obtain the new cross point B.

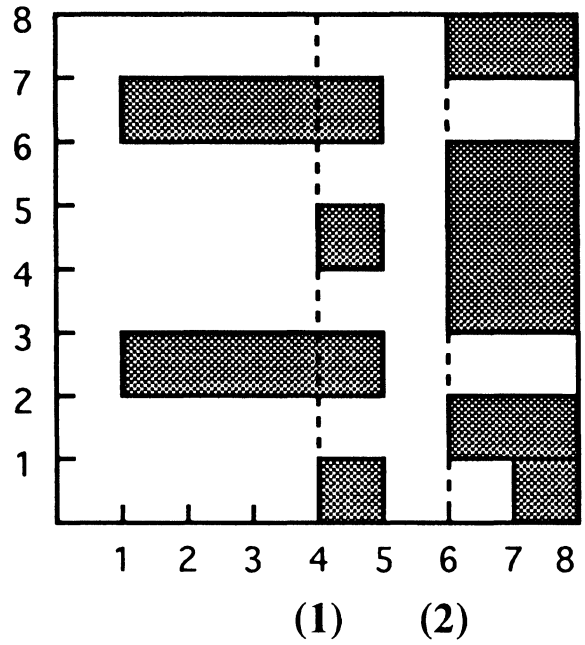

FIGURE 6 Using the binary search method to obtain the correct position (2) from the initial position (1) for the vertical dividing line. The last position is located so that the dividing line will divide the specified region into two subregions containing the same number of objects.

The following procedures written in C pseudo code are to translate the original MSQT into a balanced MSQT (BMSQT).

The function Translate() initiates balancing of the original quad tree. Balanced_split() coordinates the splitting of subregions that contain more than T_N objects. Generate_subquad() is the function that makes use of the Cut_region_vertically() and Cut_region_horizontally() procedures to determine the positions of the vertical and horizontal division lines.

Translate(QT_Root)

{This procedure translates an MS quad tree pointed to by the "QT_Root" pointer into a BMS quad tree pointed to by the "BQT_Root" pointer.}
BEGIN
if (QT_Root == NULL) return;
{Invalid input, exit and end this procedure.}

BQT_Root = Initialize(QT_Root);
{Initialize the root node of the BMS quad tree by referring to the MS quad tree input pointer. This is to pass layout plane boundary information to the BMS quad tree, and to define the root node, BQT_Root, as a non-leaf node.}

Set the whole "BQT_Root" boundary to be a "Window";

Object_reference_list = Region_query(Window, QT_Root);
{The "Object_reference_list" points to a list of object reference nodes. The "Region_query()" is a fundamental function of the MSQT[11] used to find those objects intersected by the specified "Window".}

Unmark(Object_reference_list); {Unmark the reported objects.}
Balanced_split(BQT_Root,Object_reference_list);
{Balanced_split() is the main procedure to construct a BMS quad tree. It is executed recursively.}
END {end of Translate()}

Balanced_split(BQT_node,Object_reference_list)
{The Balanced_split() is the main procedure used to construct the BMS quad tree, where the "BQT_node" points to the current node being split.}
BEGIN
int i;

if (the region size of "BQT_node" is not splittable) return;

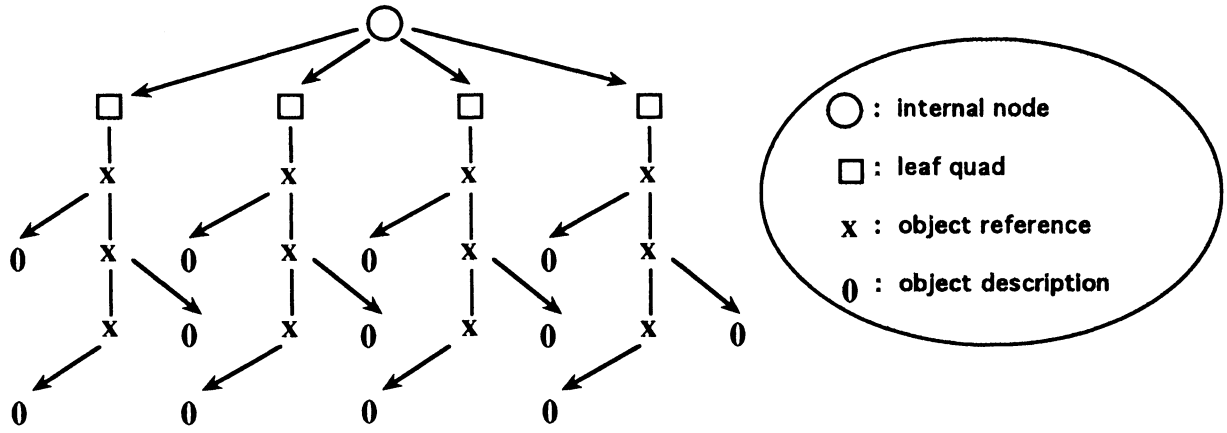

FIGURE 7 The data structure of the BMSQT for the same mask layout as shown in Figure 1(a).

```
  {Avoid indefinitely splitting quadrants that contain
   group of overlapping nested objects.}

  Generate_subquad(BQT_node);
  {Generate four subquadrants pointed to by the current
   quadrant, "BQT_node".}

  while (Object_reference_list != NULL)
  DO

  for (i = 0; i < 4; i++)
  BEGIN
  if (Object_reference_list->object intersects BQT
  _node->subquad[i])
   BEGIN
    Add this specified object to the object reference list
    of "BQT_node->subquad[i]";
    (BQT_node->subquad[i])->use ++;
    END {end of if}
END {end of for}

  Delete_node = Object_reference_list;
  {The Delete_node is a pointer pointing to the node to
   be deleted.}
  Object_reference_list = Object_reference_list->next-
  _node;
  Free(Delete_node); {Free and reallocate memory}
END {end of while}

  for (i = 0; i < 4; i++)
  if (BQT_node->subquad[i]->use > T_N)
  BEGIN
   BQT_node->subquad[i]->use = -1;
   Object_reference_list = BQT_node->subquad[i]-
   >object_reference_list;
   BQT_node->subquad[i]->object_reference_list =
   NULL;
   Balanced_split(BQT_node->subquad[i], Object_ref-
   erence_list);
   {recursion}
  END {end of for}
END {end of Balanced_split()}

Generate_subquad(BQT_node)
{Generate four subquadrants pointed to by the current
quadrant, "BQT_node".}
BEGIN
  int X, Y, i, j;

  X = Cut_region_vertically(the region defined by
  "BQT_node");

  {Apply the Cut_region_vertically() function to deter-
   mine the vertical dividing line, X = the coordinate
   returned by the Cut_region_vertically() function.}

  Y = Cut_region_horizontally(the region defined by
  "BQT_node");
  {Apply the Cut_region_horizontally() function to de-
   termine the horizontal dividing line, Y = the coordi-
   nate returned by the Cut_region_horizontally() func-
   tion.}

  for (i = 0; i < 4; i++)
  BEGIN
   Allocate a location for "BQT_node->subquad[i]";
   BQT_node->subquad[i]->use = 0;
   BQT_node->subquad[i]->object_reference_list =
   NULL;
   for (j = 0; j < 4; j++) BQT_node->subquad[i]-
   >subquad[j] = NULL;
   Define the "BQT_node->subquad[i]"boundary;
   {The assigning work is shown in Figure 8.}
  END {end of for}
END {end of Generate_subquad()}

Cut_region_vertically(region)
{This function determines the vertical dividing line in the
 specified region according to whether the subregions
 contain approximately the same number of objects.}
BEGIN
  int      Lower_bound,Upper_bound,Middle,
           Count1,Count2;
  BOOL     Balanced;

  Lower_bound = left boundary of "region";
  Upper_bound = right boundary of "region";
```

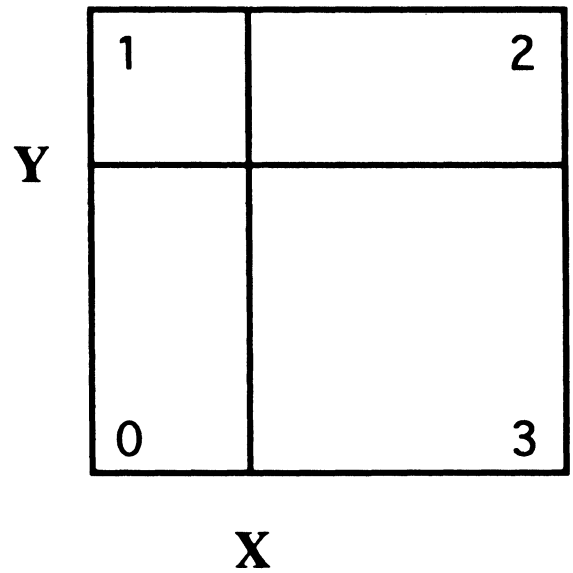

FIGURE 8 The numbered rectangular regions stand for the four subquadrants whose areas are defined by the two dividing lines and the boundaries of the specified quadrant.

```
Balanced = FALSE;
{Use the binary search method to obtain where the
 dividing line should be placed.}

REPEAT
Middle = Lower_bound + (Upper_bound − Lower-
_bound + 1) / 2;

Count1 = Region_objects(the left subregion-
,QT_Root);
Count2 = Region_objects(the right subregion-
,QT_Root);
{The Region_objects() issues a region query[12] to the
 MS quad tree pointed to by "QT_Root". With these
 two subregions generated by the dividing line, X =
 Middle, we will get the number of objects contained
 in each subregion. Here, Count1 is the number of
 objects contained in the left subregion and Count2 is
 the number of objects contained in the right subre-
 gion.}

if (Count1 == Count2) Balanced = TRUE;
else if (Count1 > Count2) Upper_bound = Middle;
   {Adjust and shift the dividing line to the left.}
   else if (Count1 < Count2) Lower_bound = Middle;
   {Adjust and shift the dividing line to the right.}
if ((Upper_bound—Lower_bound) < 2) balanced =
TRUE;
{This conditional statement acknowledge that we can
 not always divide the specified region into two
 subregions containing exactly the same number of
 objects in the vertical direction. Sometimes, the
 number of objects will differ by one.}
UNTIL (Balanced == TRUE);

 return (Middle);
END {end of Cut_region_vertically()}
```

Cut_region_horizontally(region)

{Similar to Cut_region_vertically(), this function determines placement of the horizontal dividing line in a specified region. Please refer to Cut_region_vertically() for details.}

5 COMPLEXITY ANALYSIS

5.1 Descriptive Algorithms with Complexity Analysis

Before describing the generic algorithm for translating the Original Quad Tree (OQT) into our Balanced Quad Tree (BQT), we first present the algorithm, Cut-region-vertically(), for generating vertical(horizontal) division lines. It partitions any given rectangular region containing overlapping or non-overlapping layout rectangles into regionLeft and regionRight, such that each contains exactly or approximately the same number of layout rectangles as the other.

Cut-region-vertically(OQT, region)

{

Step 1

Heuristic binary search for generating regionLeft and regionRight such that

$|Ln - Lr| \leq Rd$

where

$Rd = \lfloor T_N/10 \rfloor + 3$;/*$T_N$ = threshold value of OQT*/

Ln = Query(OQT, regionLeft);

Lr = Query(OQT, regionRight);

and region = vertically disjoint union of regionLeft and regionRight;

Step 2

vertical-division-line = vertical boundary line between regionLeft and regionRight;

Step 3

Return (vertical-division-line);

}

Note that the time complexity for the heuristic binary search shown above depends on the horizontal length (or vertical width) of the region, the distribution of layout rectangles, and our heuristics. Without heuristics and considering the layout distribution, this search needs a time of O(log M), where M is the horizontal length (or vertical width) of the region. M is independent of N, the total number of layout rectangles. With our heuristics in considering the distribution of a given layout, this search can be done in constant time. As a result, the time complexity of Cut-region-vertically() is only influenced by the time complexity of Region Query in OQT. For the worst case, the OQT may be a skewed tree. Therefore, the time complexity of Cut-region-vertically() as well as that of Region Query in OQT is O(N)

Generally, the translation algorithm, Translate(), from OQT to BQT recursively splits each partitioned region into four balanced subregions, region1, region2, region3 and region4, according to the vertical-division-line and horizontal-division-line obtained from Cut-region-vertically() and Cut-region-horizontally(). These four subregions should contain nearly equal numbers of rectangles. During the split operation, BOT tree nodes are built from OQT by referring to the calculated vertical-division-line and horizontal-division-line. For those regions requiring no further splitting, the corresponding object reference nodes in OQT are transferred to constructing BQT one by one. BOT's tree-building operation as illustrated below, is separately and recursively completed by the

functions of Build-tree-node-of-BQT() and Build-object-reference-node-of-BQT().

```
Translate(OQT)
{
  region = Find-whole-layout-region(OQT);
  BQT = Balanced-split(OQT, region);
  Return = BQT;
}

Balanced-split(OQT, region)
{
  dividing = Check-whether-further-division(OQT, region);
  if (dividing = = YES)
    {
    vertical-division-line = Cut-region-vertically (OQT, region);
    horizontal-division-line = Cut-region-horizontally (OQT, region);
    BQT = Build-tree-node-of-BQT(BQT, vertical-division-line, horizontal-division-line);
    region1, region2, region3, region4 = Generate-subquad(region, vertical-division-line, horizontal-division-line);
    Balanced-split(OQT, region1);
    Balanced-split(OQT, region2);
    Balanced-split(OQT, region3);
    Balanced-split(OQT, region4);
  }
 else /*dividing = NO */
  {
    BQT = Build-object-reference-node-of-BQT (OQT, region);
  }
 Return(BQT);
}
```

In Translate() algorithm above, the final time complexity depends on Find-whole-layout-region() and Balanced-split(). The Find-whole-layout-region() function can easily be done by Region Query in OQT, and its time complexity should be in O(N), for the same reason discussed above.

In viewing the algorithm of Balanced-split(), because that the functions of Build-tree-node-of-BQT() and Generate-subquad() can be completed by quadrant operation in constant time, hence actually, the time complexity of Balanced-split() should be determined by Build-object-reference-node-of-BQT(), Cut-region-vertically(horizontally)(), and the recursive operations of Balanced-split() itself. Here, we have already known that the time complexity of the function Cut-region-vertically(horizontally)() is in O(N). In addition, the time complexity of Build-object-reference-node-of-BQT() is linearly dependent of the number of object reference nodes for each partitioned region, which usually is much less than N. From this, we obtain that the time complexity of Balanced-split() should be in O(N * no. of recursive operations). Due to the contribution of balanced partition for each Cut-region-vertically(horizontally)(), the final tree height of BQT as well as the number of recursive Balanced-split() operations is in O(log N). Consequently, the time complexity of Balanced-split() is in O(N log N). This concludes that the time complexity of our proposed translating algorithm from OQT to BQT, Translate(), is in O(N log N). The reconstructed balanced quad tree, BQT, guarantees the time complexity of Region Query in O(log N) which is much better than the original time complexity of Region Query in O(N) for OQT.

5.2 Worst Case for Tree Height Analysis

In the following analysis, we will show that the height of the proposed balanced quad tree is limited to $\log_2 N$, where N is the total number of objects in the plane. In the construction of an MSQT or QLQT, any quadrant containing more than T_N objects is recursively split into four subquadrants. Consider the root quadrant and four subquadrants shown in Figure 9, and let a, b, c and d be the number of objects in the subquadrants. The dividing line in either the vertical or horizontal direction divides the entire region into two parts, each of which has an equal number of objects. Therefore, we have

$$a + b = c + d \text{ (in the vertical division)}, \quad (4.1)$$

and

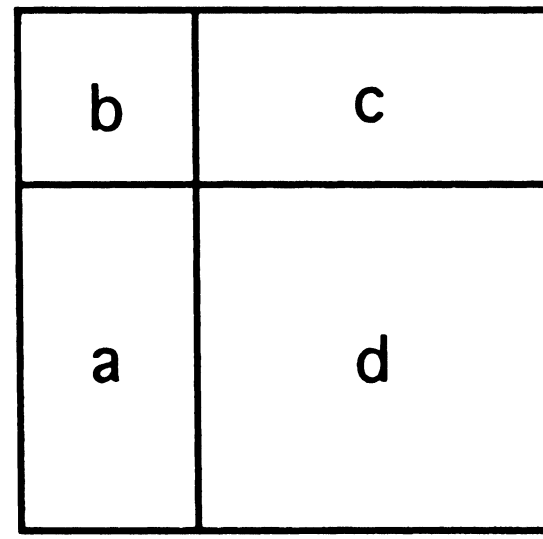

FIGURE 9 The variables a,b,c, and d representing the number of objects in the corresponding subquadrants.

$$a + d = b + c \text{ (in the horizontal division).} \quad (4.2)$$

By combining these two equations, we find that a is equal to c and that b is equal to d. For the worst case, shown in Figure 10,

$$a = c = 0 \text{ and } b = d = N/2 \quad (4.3)$$

$$(\text{or } a = c = N/2 \text{ and } b = d = 0). \quad (4.4)$$

Next, the quadrants with N/2 objects are recursively subdivided. The worst case may occur in every recursive subdivision as the example shown in Figure 11. However, in the best case, the two orthogonal divisions will divide the entire region into four parts each containing exactly the same number of objects, which implies the following equation:

$$a = b = c = d = N/4. \quad (4.5)$$

And every subquadrant, in the best case, will always be as described above. We then can conclude that the height of the balanced MSQT is limited to $\log_2 N$, and not $\log_4 N$. This is still preferable to the original MSQT, which can become skewed and suffer a linear time complexity in the worst case.

6 EXPERIMENTAL RESULTS

Region query is a basic operation needed for VLSI-CAD tool design. Therefore, region query speed is a valid measure for evaluating the performance of VLSI-CAD application data structure. To validate our proposed data structures, we conducted a series of experiments. The proposed balanced quad trees, the BMSQT and the BQLQT, were compared with the original MSQT and the QLQT using layouts generated by a pseudo random number generator and real circuits. Our program was written in C language and run on a SUN SPARC station SLC.

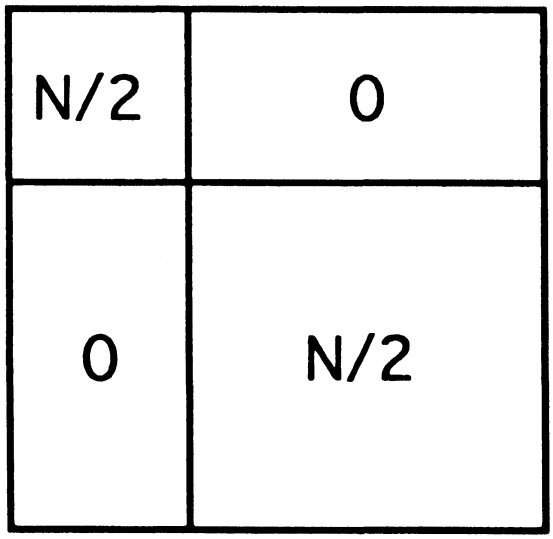

FIGURE 10 The worst case implying that a = c = 0 and b = d = N/2.

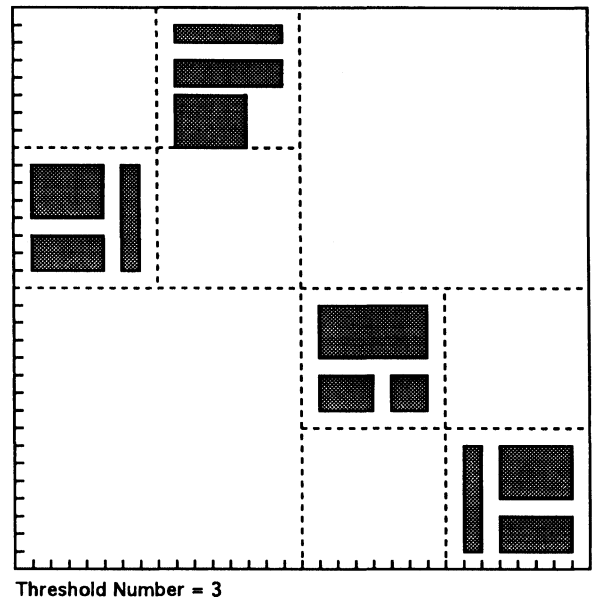

FIGURE 11 An example of the worst cases, with a group of 12 rectangles on the layout plane. (T_N = 3).

To compare the performance of the balanced quad trees with the original quad trees, we generated a large number of random windows in size ranging from 0 * 0 (a point window) to 1 * 1(a window covering the whole layout), as shown in Table 2. We then used them to query the four quad trees.

The real execution time was measured in seconds for region queries. The following terms are defined to facilitate presentation of the experimental results.

h1(h2): the average height for all the leaf quadrants in the original quad tree(the balanced quad tree).

TABLE 2
The Number of Queries Based on Random Windows Ranging in Size From 0 * 0 to 1 * 1.

window size	amount of queries
1 × 1	1
2/3 × 2/3	10
1/2 × 1/2	10
1/4 × 1/4	100
1/10 × 1/10	200
1/40 × 1/40	400
0 × 0	500

t1(t2): the average execution time for region queries in the original quad tree(the balanced quad tree).

h% (treeheight improvement ratio) = (h1–h2)/ h1 * 100%.

a% (region query improvement ratio) = (t1 (MSQT) − t1(QLQT))/t1(MSQT) * 100%.

b% (region query improvement ratio) = (t1 (MSQT) − t2(BMSQT))/t1(MSQT) * 100%.

g% (region query improvement ratio) = (t1 (MSQT) − t2(BQLQT))/t1(MSQT) * 100%.

Experiments for random layouts with 1000, 10000 and 70000 objects for the MSQT, the QLQT, the BMSQT, and the BQLQT are tested and the results were shown in Tables 3, 4, 5, 6, 7, 8, 9, 10 and 11. These experimental results confirm that both the search speed and the space requirements of the balanced quad trees are superior to those of the original quad trees, because the height of a balanced quad tree is always shorter than that of the original quad tree.

Furthermore, by observing the experimental results presented by Weyten & Pauw[12] (and shown in Table 12) for the comparison of region queries in the MSQT and the QLQT, we can conclude that those excellent results were obtained that the time exhaustive method of the marking and unmarking operations in Brown's paper, which was out of date[24][14], was compared in their experiment. In fact, the marking and unmarking operations used in the original MSQT can be improved upon by increasing the values of the flags in the object description nodes associated with each visit instead of maintaining validity by marking them and unmarking them before the next query. It is then not necessary to unmark the visited objects for every region query operation. Hence, our results obtained from using the new

TABLE 3
Memory Requirements for the Data Structures of the Four Distinct Quad Trees Based on the Same Layout of 1000 Objects.

TREE TYPE	Tree nodes			Object ref. nodes			Object nodes	Space Total (bytes)		
	T_N : 5	10	20	5	10	20	5 10 20	5	10	20
MSQT	721	337	133	1300	1186	1118	1000	63240	46968	38264
QLQT	721	337	133	1300	1186	1118	1000	67892	47012	35860
BMSQT	537	329	105	1123	1060	1035	1000	54464	45640	36480
BQLQT	537	329	105	1123	1060	1035	1000	56908	45588	33740

TABLE 4
Comparison of Tree Heights for the Original and the Balanced Quad Tree Based on the Same Layout of 1000 Objects.

T_N	Original quad trees (MSQT or QLQT) h1 (average heights for leafquads)	Balanced quad trees (BMSQT or BQLQT) h2 (average heights for leafquads)	h%
5	4.741220	4.486352	5.4%
10	4.007905	3.987854	0.5%
20	3.480000	3.253165	6.5%

TABLE 5
Memory Requirements for the Data Structures of the Four Distinct Quad Trees Based on the Same Layout of 10000 Objects.

TREE TYPE	Tree nodes			Object ref. nodes			Object nodes	Space Total (bytes)		
	T_N : 10	20	30	10	20	30	10 20 30	10	20	30
MSQT	3781	1401	1329	12909	11631	11573	10000	494512	389088	385744
QLQT	3781	1401	1329	12909	11631	11573	10000	499884	365900	361692
BMSQT	3369	1365	1125	11481	11123	10856	10000	466608	383584	371848
BQLQT	3369	1365	1125	11481	11123	10856	10000	467036	359964	345348

TABLE 6
Comparison of Tree Heights for the Original and the Balanced Quad Tree Based on the Same Layout of 10000 Objects.

T_N	Original quad trees (MSQT or QLQT) h1 (average heights for leafquads)	Balanced quad trees (BMSQT or BQLQT) h2 (average heights for leafquads)	h%
10	5.856135	5.798971	1.0%
20	5.034253	5.000000	0.7%
30	4.990973	4.928910	1.3%

TABLE 7
Memory Requirements for the Data Structures of the Four Distinct Quad Trees Based on the Same Layout of 70000 Objects.

TREE TYPE	Tree nodes			Object ref. nodes			Object nodes			Space Total (bytes)		
	TN : 20	40	60	20	40	60	20	40	60	20	40	60
MSQT	9433	5461	4937	72975	72216	72105		70000		2641120	2476168	2454320
QLQT	9433	5461	4937	72975	72216	72105		70000		2474316	2261700	2233564
BMSQT	7261	5013	4025	71076	70446	70421		70000		2541288	2444088	2404368
BQLQT	7261	5013	4025	71076	70446	70421		70000		2348228	2224244	2172668

method in marking and unmarking operations do speed up region queries in the MSQT.

Our results show that the improvement is more striking for large windows. Analyzing the value of **b%(g%) /h%** we find it grows larger as window size increases, which means larger windows further enhance improvement and make it more apparent to us. On the other hand, because smaller windows cover fewer objects, a small window then visits a small portion of the objects in the entire layout. As a result, less improvement in query operations between the balanced quad tree and the original one is apparent. By the way, the speed of the

TABLE 8
Comparison of Tree Heights for the Original and the Balanced Quad Tree Based on the Same Layout of 70000 Objects.

TN	Original quad trees (MSQT or QLQT) h1 (average heights for leafquads)	Balanced quad trees (BMSQT or BQLQT) h2 (average heights for leafquads)	h%
20	6.561413	6.330518	3.5%
40	6.000000	5.992553	0.1%
60	5.964623	5.881086	1.4%

TABLE 9
Time for Window Operations in Seconds for the Random Layout of 1000 Objects.

(a) TN = 5

Window size	MSQT	QLQT	BMSQT	BQLQT	α%	β%	γ%	β%/h%	γ%/h%
1 × 1	0.018500	0.017833	0.015067	0.014267	3.6%	18.6%	22.9%	3.452	4.256
2/3 × 2/3	0.009323	0.009090	0.007493	0.007127	2.5%	19.6%	23.6%	3.651	4.382
1/2 × 1/2	0.005553	0.005393	0.004647	0.004390	2.9%	16.3%	20.9%	3.037	3.896
1/4 × 1/4	0.001673	0.001673	0.001393	0.001348	0%	16.8%	19.4%	3.117	3.614
1/10 × 1/10	0.000537	0.000553	0.000469	0.000463	−3.0%	12.7%	13.8%	2.355	2.563
1/40 × 1/40	0.000225	0.000236	0.000212	0.000213	−4.9%	6.0%	5.3%	1.124	0.992
0 × 0	0.000173	0.000178	0.000167	0.000165	−2.9%	3.3%	4.6%	0.614	0.860

(b) TN = 10

Window size	MSQT	QLQT	BMSQT	BQLQT	α%	β%	γ%	β%/h%	γ%/h%
1 × 1	0.012033	0.011033	0.012000	0.011033	8.3%	0.3%	8.3%	0.554	16.611
2/3 × 2/3	0.006463	0.005967	0.006190	0.005793	7.7%	4.2%	10.4%	8.453	20.721
1/2 × 1/2	0.003973	0.003680	0.003730	0.003523	7.4%	6.1%	11.3%	12.241	22.639
1/4 × 1/4	0.001277	0.001229	0.001197	0.001159	3.8%	6.2%	9.2%	12.470	18.470
1/10 × 1/10	0.000413	0.000424	0.000387	0.000392	−2.7%	6.2%	5.1%	12.403	10.163
1/40 × 1/40	0.000199	0.000210	0.000199	0.000203	−5.5%	0%	−2.0%	0.047	−4.018
0 × 0	0.000160	0.000167	0.000159	0.000160	−4.4%	0.7%	0%	1.457	0

(c) TN = 20

Window size	MSQT	QLQT	BMSQT	BQLQT	α%	β%	γ%	β%/h%	γ%/h%
1 × 1	0.009300	0.008033	0.008800	0.007767	13.6%	5.4%	16.5%	0.825	2.529
2/3 × 2/3	0.005083	0.004553	0.004633	0.004240	10.4%	8.9%	16.6%	1.358	2.544
1/2 × 1/2	0.003227	0.002923	0.002960	0.002760	9.4%	8.3%	14.5%	1.268	2.220
1/4 × 1/4	0.001047	0.001010	0.000953	0.000919	3.5%	9.0%	12.2%	1.378	1.876
1/10 × 1/10	0.000368	0.000384	0.000350	0.000356	−4.3%	5.0%	3.3%	0.768	0.500
1/40 × 1/40	0.000184	0.000196	0.000178	0.000187	−6.5%	2.9%	−1.6%	0.449	−0.250
0 × 0	0.000148	0.000160	0.000144	0.000153	−8.1%	2.2%	−3.4%	0.343	−0.518

TABLE 10
Time for Window Operations in Seconds for the Random Layout of 10000 Objects.

(a) T_N = 10									
Window size	MSQT	QLQT	BMSQT	BQLQT	α%	β%	γ%	β%/h%	γ%/h%
1 × 1	0.151667	0.136000	0.137000	0.124000	10.3%	9.7%	18.2%	9.907	18.689
2/3 × 2/3	0.073117	0.065017	0.066100	0.059317	11.1%	9.6%	18.9%	9.831	19.336
1/2 × 1/2	0.042650	0.038000	0.038100	0.034250	10.9%	10.7%	19.7%	10.929	20.177
1/4 × 1/4	0.010764	0.009736	0.009687	0.008762	9.6%	10.0%	18.6%	10.247	19.054
1/10 × 1/10	0.002105	0.001963	0.001935	0.001810	6.7%	8.1%	14.0%	8.275	14.357
1/40 × 1/40	0.000443	0.000444	0.000431	0.000423	−0.2%	2.7%	4.5%	2.730	4.625
0 × 0	0.000235	0.000238	0.000235	0.000235	−1.3%	0.1%	0%	0.116	0
(b) T_N = 20									
Window size	MSQT	QLQT	BMSQT	BQLQT	α%	β%	γ%	β%/h%	γ%/h%
1 × 1	0.112833	0.096500	0.109333	0.093667	14.5%	3.1%	17.0%	4.559	24.965
2/3 × 2/3	0.054867	0.047233	0.053417	0.046300	13.9%	2.6%	15.6%	3.884	22.948
1/2 × 1/2	0.032067	0.027667	0.030983	0.026867	13.7%	3.4%	16.2%	4.965	23.833
1/4 × 1/4	0.008140	0.007138	0.007856	0.006902	12.3%	3.5%	15.2%	5.131	22.353
1/10 × 1/10	0.001660	0.001536	0.001616	0.001477	7.5%	2.6%	11.0%	3.872	16.202
1/40 × 1/40	0.000390	0.000395	0.000386	0.000382	−1.3%	1.2%	2.1%	1.728	3.015
0 × 0	0.000224	0.000232	0.000222	0.000222	−3.6%	0.7%	0.9%	1.094	1.312
(c) T_N = 30									
Window size	MSQT	QLQT	BMSQT	BQLQT	α%	β%	γ%	β%/h%	γ%/h%
1 × 1	0.111833	0.095167	0.104667	0.090500	14.9%	6.4%	19.1%	5.153	15.340
2/3 × 2/3	0.054400	0.047067	0.051750	0.045433	13.5%	4.9%	16.5%	3.917	13.256
1/2 × 1/2	0.031733	0.027650	0.029800	0.026350	12.9%	6.1%	17.0%	4.899	13.642
1/4 × 1/4	0.008072	0.007153	0.007645	0.006841	11.4%	5.3%	15.3%	4.251	12.264
1/10 × 1/10	0.001650	0.001534	0.001601	0.001497	7.0%	3.0%	9.3%	2.415	7.457
1/40 × 1/40	0.000388	0.000400	0.000388	0.000391	−3.1%	0%	−0.8%	0.027	−0.622
0 × 0	0.000222	0.000234	0.000226	0.000229	−5.4%	−1.9%	−3.2%	−1.546	−2.536

QLQT with small windows is worse than that of the MSQT [12][13] because it involves too many extra tests concerning the types of quad lists.

Our results are encouraging, even though these experiments were based on random layouts with a nearly uniform distribution. When uneven layouts were considered, the performance advantage resulting from our method is considerably greater. Table 10 shows that the query performance of BMSQT and BQLQT is better than that of the QLQT and of the MSQT. Our method is a general technique for improving all 2D spatial data structures that use region query operations. From this perspective, the improvement to all of the 2D spatial data structures provided by our technique appears much better than the contribution of the QLQT[12] to the MSQT[11].

Considering the real circuit shown in Figure 12-13, this mask layout uses 1006 rectangles. It takes about 0.4 seconds to build up the original QLQT, and 0.6 seconds to translate the original QLQT into our balanced QLQT. Additional processing, such as compaction, design-rule checking and circuit extraction, usually requires hundreds or thousands of Region Queries but only uses one period of tree building and translation. The proposed translation algorithm and the balanced quad trees themselves were proven to be more worthwhile. Figure 12-13 shows that memory usage was reduced from 64 Kbytes to 46 Kbytes (or about 30%), and the average tree-height was very much improved: from 18.4 to 3.7. Note that from the tree height distribution illustrated in Figure 12, there are 88 leaf nodes with tree height= 4 and 42 leaf nodes with tree height = 3. This means, in this case, the balanced QLQT was proven to be a fully balanced quad tree.

7 CONCLUSIONS

The proposals in his paper contribute to reduction in memory usage and better query efficiency for the proposed balanced quad trees over original quad trees. The key to building the balanced quad trees described here is determining the locations of dividing lines in a specified region; that is, to adjust the cross point to a suitable position, as shown in Figure 4. The time complexity of the presented translation algorithm and that of the original quad tree building algorithm are in the same time order of O(N log N). However, the reconstructed balanced quad trees guarantee the time complexity of Region Query in O(log N), which is much better than the original time complexity of Region Query in O(N) for OQT.

TABLE 11
Time for Window Operations in Seconds for the Random Layout of 70000 Objects.

(a) $T_N = 20$

Window size	MSQT	QLQT	BMSQT	BQLQT	α%	β%	γ%	β%/h%	γ%/h%
1 × 1	0.754167	0.682333	0.707000	0.658833	9.5%	6.3%	12.6%	1.777	3.592
2/3 × 2/3	0.339650	0.313200	0.322283	0.296167	7.8%	5.1%	12.8%	1.453	3.638
1/2 × 1/2	0.193700	0.178433	0.182600	0.167800	7.9%	5.7%	13.4%	1.628	3.800
1/4 × 1/4	0.049832	0.047192	0.047424	0.044255	5.3%	4.8%	11.2%	1.373	3.180
1/10 × 1/10	0.008360	0.008206	0.007931	0.007669	1.8%	5.1%	8.3%	1.456	2.349
1/40 × 1/40	0.000978	0.001030	0.000948	0.000972	−5.3%	3.1%	0.6%	0.866	0.174
0 × 0	0.000281	0.000299	0.000279	0.000292	−6.4%	0.7%	−3.9%	0.186	−1.112

(b) $T_N = 40$

Window size	MSQT	QLQT	BMSQT	BQLQT	α%	β%	γ%	β%/h%	γ%/h%
1 × 1	0.585833	0.533167	0.571667	0.520333	9.0%	2.4%	11.2%	19.483	90.094
2/3 × 2/3	0.269900	0.248117	0.264000	0.242650	8.1%	2.2%	10.1%	17.612	81.356
1/2 × 1/2	0.153550	0.141383	0.150550	0.138867	7.9%	2.0%	9.6%	15.742	77.054
1/4 × 1/4	0.039287	0.036723	0.038456	0.035923	6.5%	2.1%	8.6%	17.039	68.998
1/10 × 1/10	0.006082	0.005820	0.005924	0.005657	4.3%	2.6%	7.0%	20.964	56.308
1/40 × 1/40	0.000717	0.000727	0.000731	0.000731	−1.4%	−1.9%	−2.0%	−15.251	−15.734
0 × 0	0.000222	0.000231	0.000227	0.000235	−4.1%	−2.3%	−5.9%	−18.433	−47.187

(c) $T_N = 60$

Window size	MSQT	QLQT	BMSQT	BQLQT	α%	β%	γ%	β%/h%	γ%/h%
1 × 1	0.676333	0.611500	0.664167	0.594167	9.6%	1.8%	12.1%	1.284	8.675
2/3 × 2/3	0.311900	0.282500	0.306483	0.275417	9.4%	1.7%	11.7%	1.240	8.352
1/2 × 1/2	0.177383	0.161767	0.172433	0.157750	8.8%	2.8%	11.1%	1.993	7.903
1/4 × 1/4	0.046558	0.043028	0.045572	0.042198	7.6%	2.1%	9.4%	1.513	6.687
1/10 × 1/10	0.007820	0.007516	0.007665	0.007428	3.9%	2.0%	5.0%	1.416	3.579
1/40 × 1/40	0.000952	0.000997	0.000956	0.000990	−4.7%	−0.4%	−4.0%	−0.274	−2.850
0 × 0	0.000286	0.000309	0.000292	0.000310	−8.0%	−2.0%	−8.4%	−1.449	−5.992

The experimental results shown in Figure 12-13 and Table 9-12 prove that the balanced quad trees improve all 2D spatial data structures that use region query operations even more than the QLQT improves on the MSQT. For a heavily non-uniformly distributed layout, the improvement in performance provided by our method will be especially significant. From the real circuit shown in Figure 12-13, we obtained about 30% reduction in memory usage, and an average tree height reduction from 18.4 to 3.7. Moreover, for most of real circuits, our balanced QLQT was proven to be a fully balanced quad tree as indicated by the tree height distribution on the message panel of Figure 12-13.

This method also preserves all the advantages of the original quad trees. The proposed data structures speed up search and window query operations significantly and hence should be of great value in the design of VLSI layout tools.

In our research work, we have developed a layout editor[25], a complementary DRC[26], a global router with minimum tile partition[27], and two different layout compactors[28] based on the presented BQLQT.

TABLE 12
The Experimental Results presented by Weyten & Pauw, the considered Layout Contains 7351 Objects.
Window Operation Time in Seconds ($T_N = 30$)

Window size	Brown	QLQT	Brown/QLQT
0 × 0	.00264	.00143	1.85
1/40 × 1/40	.00648	.00258	2.51
1/10 × 1/10	.03279	.00901	3.64
1/4 × 1/4	.15028	.03515	4.28
1/2 × 1/2	.54568	.11798	4.63
1 × 1	1.97200	.40100	4.92

Acknowledgement

This work was supported in part by NSC82-0404-E004-129. In preparing this manuscript, many thanks to Mr. L. D. Jang from National Chiao Tung University of Taiwan, ROC, and Mr. P. W. Shew from National University of Singapore.

References

[1] J. K Ousterhout, "Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools," IEEE Trans. Computer Aided Design, Vol. CAD-3, pp. 87–100, Jan. 1984.

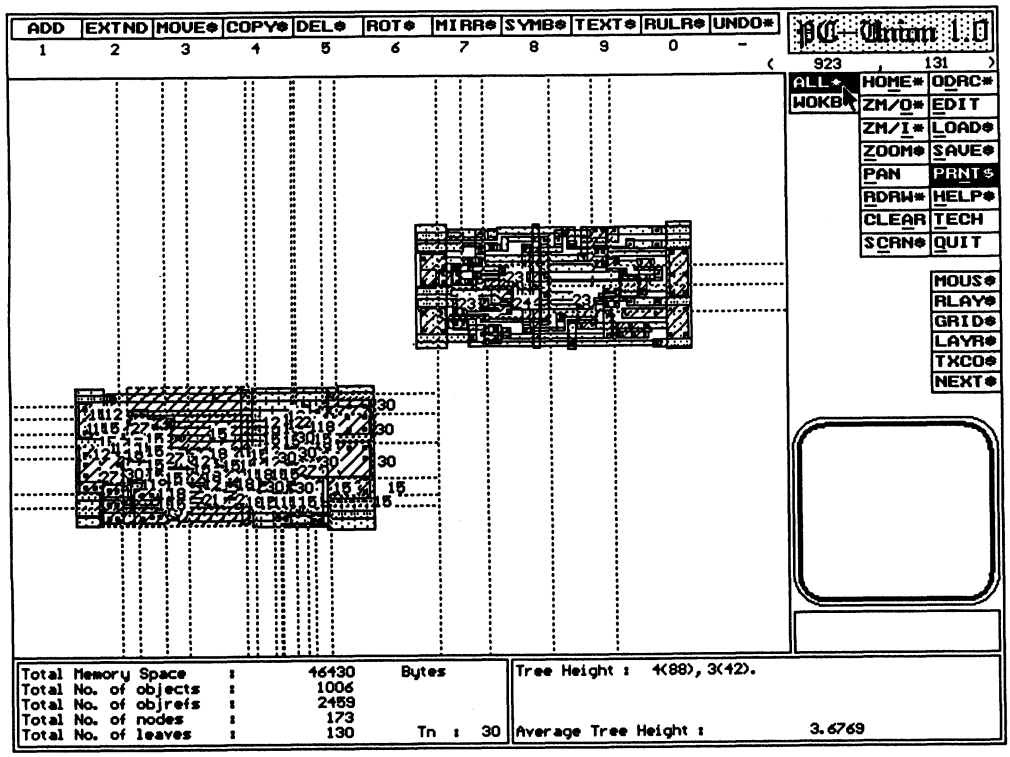

FIGURE 12 The Balanced QLQT (for real circuit shown in Figure 13) with average tree height = 3.7 and memory usage = 46 Kbytes.

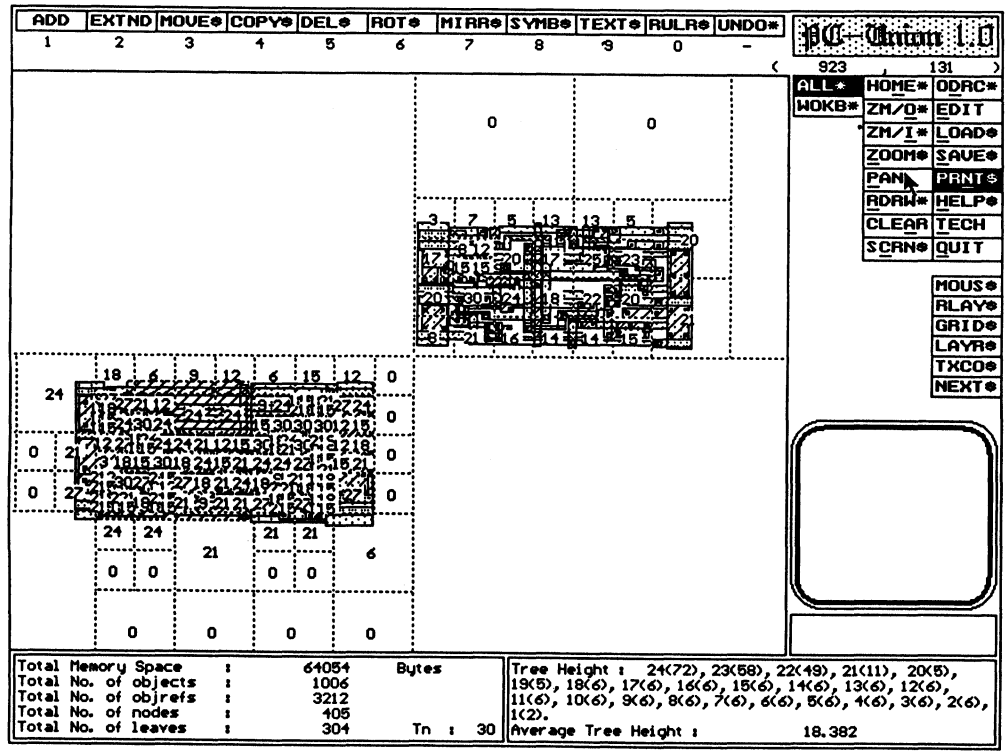

FIGURE 13 Real layout circuit with 1006 rectangle. The Original QLQT with average tree height = 18.4 and memory usage = 64 Kbytes.

[2] J. Bentley, "Multidimensional Binary Search Tree Used for Associative Searching, Communications ACM, Vol. 18, No. 9, pp. 509–517, Sep. 1975.

[3] U. Lauther, "4-Dimensional Binary Search Trees as A Means to Speed Up Associative Searches in Design Rule Verification of Integrated Circuits," J. Design Automation and Fault Tolerant Computing, Vol. 2, No. 3, pp. 241–247, Jul., 1978.

[4] Rosenberg, J.B., "Geographical Data Structures Compared: A Study of Data Structures Supporting Region Queries," IEEE Trans. on CAD/ICS, CAD-4, pp. 53–67, 1985.

[5] A. Pitaksanonkul, S. Thanawastien and C. Lursinsap, "Bisection Trees and Half Quad Trees: Memory and Time Efficient Data Structures for VLSI Layout Editors," INTEGRATION, the VLSI Journal 8(1989), pp. 285–300, 1989.

[6] A. Pitaksanonkul, S. Thanawastien, and C. Lursinsap, "Comparisons of Quad Trees and 4-D Trees: New Results," IEEE Trans. on CAD/ICS, Vol. 8, No. 11, pp. 1157–1164, 1989.

[7] H. Samet, "The Quad Tree and Related Hierarchical Data Structures," Computing Surveys, Vol. 16, pp. 187–260, June, 1984.

[8] R. Finkel and J. Bentley, "Quad Trees - A Data Structure for Retrievals on Composite Keys," Acta Informatica, Vol. 4, pp. 1–9, 1974.

[9] G. Kedem, "The Quad-CIF Tree: A Data Structure for Hierarchical Online Algorithm," in Proc. 19th IEEE Design Automation Conf., pp. 325–357, Jun., 1982.

[10] S. K. Nandy and I. V. Ramakrishnan, "Dual Quad Tree Representation for VLSI Designs," IEEE/ACM 23rd Design Automation Conf., pp. 663–666, 1986.

[11] R. L. Brown, "Multiple Storage Quad Tree: A Simpler Faster Alternative to Bisector List Quad Trees," IEEE Trans. on CAD/ICS, CAD-5, Vol. 3, pp. 413–419, 1986.

[12] L. Weyten and W. de Pauw, "Quad List Quad Trees: A Geometrical Data Structure with Improved Performance for Large Region Queries," IEEE Trans. on CAD/ICS, Vol. 8, No. 3, pp. 229–233, 1989.

[13] L. Weyten and W. de Pauw, "Performance Prediction for Adaptive Quad Tree Graphical Data Structures," IEEE Trans. on CAD/ICS, Vol. 8, No. 11, pp. 1218–1222, 1989.

[14] P. Y. Hsiao and W. S. Feng, "An Edge Oriented Compaction Scheme Based on Multiple Storage Quad Tree," IEEE Symp. Circuits & Systems, pp. 2435–2438, 1988.

[15] P. Y. Hsiao and W. S. Feng, "Using a Multiple Storage Quad Tree on a Hierarchical VLSI Compaction Scheme," IEEE Trans. on CAD/ICAS, Vol. CAD-9, No. 5, pp. 522–536, 1990.

[16] S. K. Nandy and L. M. Patnaik, "Linear Time Geometrical Design Rule Checker Based on Quadtree Representation of VLSI Mask Layouts," Computer Aided Design, Vol. 18, No. 7, pp. 380–388, 1986.

[17] W. Schiele, "Automatic Design Rule Adaptation of Leaf Cell Layouts," INTEGRATION, VLSI J., No. 3, pp. 93–112, 1985.

[18] S. K. Nandy and L. M. Patnaik, "Algorithm for Incremental Compaction of Geometrical Layouts," Computer Aided Design, Vol. 19, No. 15, pp. 257–265, 1987.

[19] W. S. Scott and J. K. Ousterhout, "Plowing: Interactive Stretching and Compaction in Magic," in Proc. 21st IEEE Design Automation Conf. pp. 166–172, 1984.

[20] A. Margarino, A. Romano, A. De Gloria, F. Curatelli, and P. Antognetti, "A Tile-Expansion Router," IEEE Trans. on CAD/ICAS, Vol. CAD-6, pp. 507–517, 1987.

[21] Y. L. Lin and Y. C. Hsu, "A New Algorithm for Tile Generation," INTEGRATION, the VLSI Journal, No. 9, pp. 259–269, 1990.

[22] P. Y. Hsiao and C. C. Tsai, "A New Plane-Sweep Algorithm Based on Spatial Data Structures for OverLapped Rectangles in 2-D plane," IEEE 14th Annual Int. Computer Software & Applications Conf., pp. 347–352, 1990.

[23] C. C. Tsai, S. J. Chen, P. Y. Hsiao, and W. S. Feng, "A New Iterative Construction Approach to Routing with Compacted Area," IEE Proceedings - E Computer & Digital Techniques, Vol.138, No.1,pp.57–71, 1991.

[24] P. Y. Hsiao and W. S. Feng, "With M.S. Quad Tree on the Constraint Graph Compaction of the VLSI/CAD Large-Cell Layout-Editor," Int. Symp. on VLSI/TSA, May 1987, pp. 297–301.

[25] P.Y. Hsiao and J.T. Yan, "PC-UNION: A Low Cost Layout Tool for Consumer IC Design," Int Symp. on IC Design, Manufacture, and Applications, pp.452–457, 1991.

[26] P.Y. Hsiao and J.T. Yan, "Designing a Complementary Design Rule Checker Based on Binary Balanced Quad List Quad Tree," IEE Proceedings-E Computers and Digital Techniques, Vol.139, No.4, pp.311–322, 1992.

[27] P.Y. Hsiao, C. Y. Lin, and P.W. Shew, "OTP: An Optimal Tile Partition for the Space Region of Integrated Circuits Geometry," IEE Proceedings-E Computers and Digital Techniques, Vol.140, No.3, pp.145–153, 1993.

[28] P.Y. Hsiao and L. D. Jang, "Using a Balanced Quad Tree to Speed up a Hierarchical VLSI Compaction Scheme," IEEE 5th Int. Conf. on VLSI Design, pp.370–371, 1992.

[29] S.J. Su, Y.S. Kuo and J.C. Tsay, "Adaptable quad tree techniques," INTEGRATION, the VLSI Journal 15(1993), pp. 51–71, 1993.

[30] W. De Pauw, "Multitrees with Internal Storage," IEEE Trans. on CAD/ICS, Vol. 12, No. 10, pp. 1428–1436, Oct. 1993.

Biographies

PEI-YUNG HSIAO received the B.S. degree in Chemical Engineering from Tung Hai University in 1980, and the M.S. and Ph.D. degrees in Electrical Engineering form the National Taiwan University in 1987 and 1990, respectively. Since August 1990, he has been an Associate Professor in the Department of Computer and Information Science at the National Chiao Tung University, Hsinchu, Taiwan, ROC. His main research interests are VLSI-CAD, Neural Network, and Expert System Applications.